LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
Il6r
no. 293-300
cop. 2

CENTRAL CIRCULATION AND BOOKSTACKS

The person borrowing this material is responsible for its renewal or return before the **Latest Date** stamped below. **You may be charged a minimum fee of $75.00 for each non-returned or lost item.**

Theft, mutilation, or defacement of library materials can be causes for student disciplinary action. All materials owned by the University of Illinois Library are the property of the State of Illinois and are protected by Article 16B of *Illinois Criminal Law and Procedure*.

TO RENEW, CALL (217) 333-8400.

University of Illinois Library at Urbana-Champaign

JUN 2 8 1999

FEB 0 1 A.M.

When renewing by phone, write new due date below previous due date. L162

Digitized by the Internet Archive
in 2013

http://archive.org/details/tranquilarithmet296budn

10.84
Il6N
o.296
Cop. 2

Math

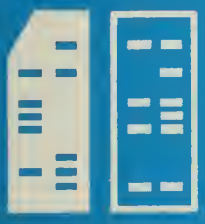
Report No. 296

TRANQUIL ARITHMETIC

by

Paul Peter Budnik

January 6, 1969



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

THE LIBRARY OF THE
... 1969
UNIVERSITY OF ILLINOIS

Report No. 296

TRANQUIL ARITHMETIC*

by

Paul Budnik

January 6, 1969

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

* This work was supported in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. US AF 30(602)4144 and submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, February, 1969.

ACKNOWLEDGMENT

I wish to thank Professor Kuck who originally conceived the general form of the TRANQUIL language and its data structures, and who contributed to TRANQUIL throughout its development, and Professor Northcote for his assistance in our implementation of TRANQUIL. A number of applications programmers contributed ideas which aided in the development of TRANQUIL.

Additional thanks goes to Norma Abel for implementing Boolean expressions and doing some additional coding and for her numerous questions which have often helped to clarify things in my own mind. I am grateful to Paul Kraska for assuming the responsibility of implementing set expressions. I wish to thank Yoichi Muraoka and Bob Wilhelmson, my colleagues on this project.

Finally, I would like to thank Mrs. Sharon Hardman for typing this thesis. Her help was essential in getting the thesis finished on time.

ABSTRACT

The assignment statements allowed in Tranquil are described with emphasis placed on writing efficient code. A description of the implementation of the assignment statement part of the compiler is given in considerable detail and is likely to be of interest mainly to those who find it necessary to correct, alter or make additions to the Tranquil Compiler.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENT. iii

ABSTRACT. iv

LIST OF FIGURES vi

1. TRANQUIL ARITHMETIC 1

1.1 Introduction 1
1.2 Writing Efficient Tranquil 1
1.3 Set Assignment Statements 3
1.4 Boolean Assignment Statements. 4

2. PASS 2 - MACHINE CODE GENERATION. 5

2.1 Pass 1 Output. 5
2.2 Overall Structure of Pass 2. 5
2.3 Assignment Statement Compilation 8
2.4 Tables Involved in Assignment Statement Analysis 11
2.5 Compiling Code From the Tables 16

3. OPTIMIZING SIMPLE ASSIGNMENT STATEMENTS 18

4. CONCLUSION. 22

APPENDIX: A SAMPLE TRANQUIL PROGRAM. 23

TRANQUIL ARITHMETIC TABLES. 24

LIST OF REFERENCES. 25

LIST OF FIGURES

Figure		Page
1.	REAL ARRAY A[**0:7, **0:7, 0:N-1]	13
2.	A set of interrelated assignment statements and their tree structure.	19

1. TRANQUIL ARITHMETIC

1.1 Introduction

In designing a compiler for ILLIAC IV, it was necessary to provide facilities for expressing parallel operations, with sufficient generality to easily express almost any type of calculation. This inevitably results in permissible constructs which cannot be efficiently compiled for ILLIAC IV. This is not in itself bad. Most programs will require a small number of operations that cannot be done efficiently on ILLIAC IV. Because these operations are essential to a much longer program which can be efficiently compiled, it is necessary that one be able to express them in Tranquil. It is, however, important that the programmer be aware of what constructs do not result in efficient code so that he only uses them when necessary. Unfortunately, in providing a sufficiently general language and algorithms for compiling it, the problem of deciding how efficiently a given statement will compile does not have a trivial solution. For these reasons, the following discussions of assignment statements is in large part concerned with writing efficient code.

1.2 Writing Efficient Tranquil

The cartesian product form of SIM control forces the compiler to make decisions about which index to vary across the PE's. The basis of this decision is an estimate of the size and density of the sets involved, and how well the particular type of set matches up with the particular index of the type of array used. For example, use of any

type set on a column of an array lying in a single PE will be very inefficient.

The comma pairwise form of SIM control allows a programmer to specify things like picking out an arbitrary diagonal of a matrix. For example:

```
FOR (I,J)  SIM ([1,2,...,100], [2,3,...,101])
     DO
        B[I] ← A[I,J];
```

For SKEWED arrays this cannot be done efficiently. Consecutive diagonal elements actually occur in every other PE. To collapse this set of elements a very large number of routes are required. Similar problems arise all the time through the use of comma linked sets, and one cannot determine how well a given construct of this type will compile without actually looking at the data structures and indexing involved.

Word sizes other than those occupying a full 64 bit word are packed by putting consecutive elements in consecutive PE's and wrapping around in the same memory address (varying PE number only) until all usable space has been filled. This allows the mode and indexing operations to work in closely similar fashion for all types of word sizes.

The choice of storage scheme is probably the single decision that can have the greatest ultimate effect on the overall efficiency of the compiled program. The straight storage scheme permits efficient access of rows or diagonals. The skewed scheme permits easy access of rows and columns. These statements can serve as very general guides as to what type we should declare a given array.

1.3 Set Assignment Statements

The set assignment statement allows one to define the contents of some set in terms of an expression involving other sets. The operands in a set expression can be any type of set, but the operator determines the type of the resulting set. The standard set operations like union intersection and relative complement yield MONOSETs. The two operators CONCAT and DELETE yield GENSETs.

The two operators ", " and × form sets of higher dimension then their operands. They correspond to the "," and × used in SIM control statements.

R = [1, 2, 3, 4, 5]

S = [2, 4, 6, 8, 10]

T = [6, 4, 6, 5, 6, 7]

U = [100, 40, 0, 13]

R UNION U is [0, 1, 2, 3, 4, 5, 13, 40, 100]

R CONCAT S is [1, 2, 3, 4, 5, 2, 4, 6, 8, 10]

T COMPL R is [6, 7]

T DELETE R is [6, 6, 6, 7]

[1, 2, 3, 4], [2, 4, 6, 8] is [[1, 2], [2, 4], [3, 6], [4, 8]]

[1, 2] × [3, 4] is [[1, 3], [1, 4], [2, 3], [2, 4]]

[1, 2] × [3, 4] , [5, 6] is [[1, 3, 5], [1, 4, 6], [2, 3, 5], [2, 4, 6]]

Most of the set operations are fairly efficient on ILLIAC IV provided the sets are large enough to make use of a substantial percentage

of the PE's. The one exception is when a conversion from a MONOSET to a GENSET is required [1].

1.4 Boolean Assignment Statements

The standard ALGOL like boolean expressions are allowed in TRANQUIL. Boolean arrays are not allowed, since sets serve the same function more efficiently. Logical operators are also allowed in arithmetic expressions, thus providing full word logical operations.

2. PASS 2 - MACHINE CODE GENERATION

2.1 Pass 1 Output

Pass 1 consists of a table driven parser [2] with scanner and a large switch branching to semantic actions which are transferred to at various points in the parse. The semantic actions construct tables and output a file of intermediate language code, which is a special representation of the original program that will be compiled into machine code by Pass 2. The format of this intermediate language code is 16 bit words, the first four bits of which contain a table number and the remaining 12 contain an address in the table. Table 0 is the operator table. Entries in all other tables are considered to be operands. Ordinarily, special symbols and arithmetic and other operators are mapped into intermediate language operators and identifiers and literals are mapped into operands.

2.2 Overall Structure of Pass 2

In designing an overall structure for Pass 2, the basic considerations were to make it flexible, highly segmented, and reasonably easy to change. The concept of actions for semantic analysis, used in Pass 1, was carried over to Pass 2. Each intermediate language operator is provided with a specific action in Pass 2. Actions are also provided for each type of operand. All these actions are contained in a procedure called EXEC2.

BRANCH ON TABLE FIELD OF IL ENTRY :

0 - operator	Branch on operator number	
	Branch on context sometimes	
1 - identifier	Branch on context	
	Branch on identifier type	
2 - literal	Branch on context	
3 - string	Branch on context	
4 - dope	Branch on context	

The intermediate language code is read sequentially through a function called NEXTILWORD. Ordinarily, this is done at the head of EXEC2 by the assignment statement NXTILWRD := NEXTILWORD. The table field of NXTILWRD is then used in a case statement. In the case of operators, table 0, another case statement branches on the operator number. (This is actually done by two nested levels of case statements.)

A method is provided for executing different actions for the same operator or operand, based on the context in which the operator or operand appears. This feature was designed to be quite general. EXEC2 has one argument: CONTEXT. Upon entry to EXEC2, the procedure CHANGE (CONTEXT) is called. This procedure is simply a case statement which allows an arbitrary number of variables to be set as a function of the value of CONTEXT. These variables are used in case statements to select which action is to be executed. As an example, identifiers can occur in an assignment statement either in subscript expressions or as ordinary operands. In the intermediate language, identifiers occurring in subscript expressions are bracketed by the operators: SUBSCRIPTEDPRIMARY and ENDOFSUBSCRIPTLIST. Thus the action for subscripted primary calls EXEC2(SUBSCRIPTEDPRIMARYCONTEXT). The action for ENDOFSUBSCRIPTLIST must exit from EXEC2. This is done by transferring to label EXIT. Then, back in the action for SUBSCRIPTEDPRIMARY we must restore the context. This is done by the define RESTORECONTEXT.

Although normally words from the ILCODE file are read at the head of EXEC2, this is not necessarily always the case. NEXTILWORD can be called from inside an action. Often, when this is done, NEXTILWORD

is called until some operator or operand not in a special set is encountered. If it is necessary to execute the action for this IL code word, this can be done by transferring to the label THISIL instead of NEXTIL, provided that the IL word is stored in the variable NXTILWRD. One can accomplish the same thing in calling EXEC2 by setting the boolean variable NOTSAMEOPERATOR to FALSE before calling EXEC2.

2.3 Assignment Statement Compilation

The use of sets, the notion of SIM, the number of different types of arithmetic and storage schemes, combined with the need to compile efficient code for a parallel machine necessitate a substantial analysis of each assignment statement. We now consider this analysis as it is carried out in Pass 2 of the compiler.

The analysis is effected in several passes over the postfix intermediate language. Consider the second assignment statement in the example in Appendix A:

$$A[I, K] \leftarrow A[I + 1, K] + B[I, K];$$

Before we even begin to generate code a decision must be made as to which index is to be processed simultaneously (i.e., across the PE's) and which is to be done sequentially. The first pass over the intermediate language determines this and also copies the intermediate language into a table to be used for future passes. When a set linked* identifier is entered in

*We say that I is set linked to II in a statement like FOR (I) SIM (II) DO.

the table, additional information provided by the set definition or declaration is also entered. In the case of I, which is linked via SIM control to II, the set is known exactly and precise information from the set definition is entered in the table. For K the compiler makes an estimate of the size and density of the set based on the upper bounds given in the declaration of KK.

In general, when operations are performed on pairs of subscripts or pairs of subscripted arrays, information about the interaction between these subscripts must be generated. For example, in the case of the subscript expression I + 1 in the example above, the addition of 1 in no way alters the size, density, or type of the set. Thus, the information provided for I will be recopied with the + operand.

After the subscript expression has been processed, a check is made to see how well the type of set resulting from the index expression will work with the particular dimension of the set involved. In the example, there are only two dimensional skewed arrays in which either columns or rows can be easily accessed in parallel. If one of the arrays were straight, then at this point it would be discovered that no set will work well for the column index, because each column is stored in a single PE. This information plus information about the set density, set size, and the array size are all combined to compute a probable efficiency; i.e., the number of PE's that will probably be on if this index were varied simultaneously. Of course, it is easy to think up cases in which the estimate will be totally wrong, but in most practical cases encountered, the estimate is reasonable. A table of these probable efficiencies

is generated for each set. If the set appears in different subscripts, then on the second occurrence the new estimate is set to the minimum of the previous and present estimates.

When the end of the assignment statement is reached, the table of probable efficiencies is sorted and the result of this determines the order in which the indicies will vary. In the example K will be the index chosen to vary across the PE's because the set II is known to be small (6 elements) and the declaration of KK holds the probability of it being fairly large. Now an outer loop must be compiled to generate sequentially the elements of II. Finally, the remainder of the statement is compiled. The effect of the code that is compiled for the example assignment statement follows.

One local data buffer location is set aside as an index to the mode words of KK. Four more locations are set aside for the base addresses of the subblocks of the arrays A and B. The first mode word for KK is loaded and the leading ones detector is used to set the first value of K. This value, plus the base address of A, plus 1, plus the indexed set 0, 1, 2, ..., 63 in the PE index registers is used to access the first column of A. In a similar manner the address for the first row of B is fetched, loaded into RGR and a route left one PE is performed. The addition is executed and the first mode word for KK is used to store the result in A. Now the same process is repeated for the next subblock of A, except that the mode pattern for KK must be ended with a word having 36 1's followed by 28 0's, because the second subblock of A is only 36

words wide. Additional complications, such as pairwise SIM control specification, small subarrays, and SIM blocks add to the complexity, but not to the substance, of the algorithm outlined above.

2.4 Tables Involved in Assignment Statement Analysis

The different types of data structures, indexing and arithmetic make the process of analyzing an assignment statement fairly complex. The initial analysis is done by constructing a number of interlinked tables. These are used to do some final analysis and ultimately compile code.

The structure of these tables is complicated and not terribly interesting. What follows is a detailed description of them for the purpose of those who find it necessary to work with them directly.

A number of processes involved in assignment statement analysis is iterative; i.e., require an unspecified number of passes over parts of the assignment statement. For this and other reasons the IL for an assignment statement is stored in an array called ARITHSTACK. Entries are made in ARITHSTACK via the macros PUSH and OPRPUSH. These provide the facility for accessing the last two operands by stacking up pointers in the top of ARITHSTACK. Thus, there are two pointers associated with ARITHSTACK. One points to the next free location from the bottom of the stack (ARITHSTACKPTR) and the other points to the next free location from the top of the stack (PTRARITHSTACKPTR). It is always possible to access the last two operators in ARITHSTACK while it is being built. They are:

ARITHSTACK [ARITHSTACK[PTRARITHSTACKPTR]]

and ARITHSTACK [ARITHSTACK[PTRARITHSTACKPTR+1]].

(Macros are provided for obtaining the two operands.)

This is necessary because, when an arithmetic operator (like +) is encountered, information must be generated which is a function of its two operands.

In the cartesian product form of SIM control, each index set is ordinarily varied independently; i.e., one index set will vary across the PE's and others will form nested loops. In the comma [3] form of SIM control, two index sets vary together or elements of both are used together across the PE's. A similar thing happens with small subarrays. Say we have 8 subarrays, $8 \times N$ each stored across the PE's. (See Figure 1.) We wish to use more than 8 PE's at a time; therefore, we must use all indicies for both the first and second subscript at once, and the indicies can be thought of as varying together.

In compiling code for an assignment statement, after it has been decided in which order to vary the index sets involved, code must be produced to separate the elements of the sets, or to provide for their availability in the PE's. Once the elements of some sets are available, it is desirable to compute any subexpression involving only these variables in the outermost loop. To do this efficiently, we keep track of all occurrences of set linked variables and link together those variables which must be varied together.

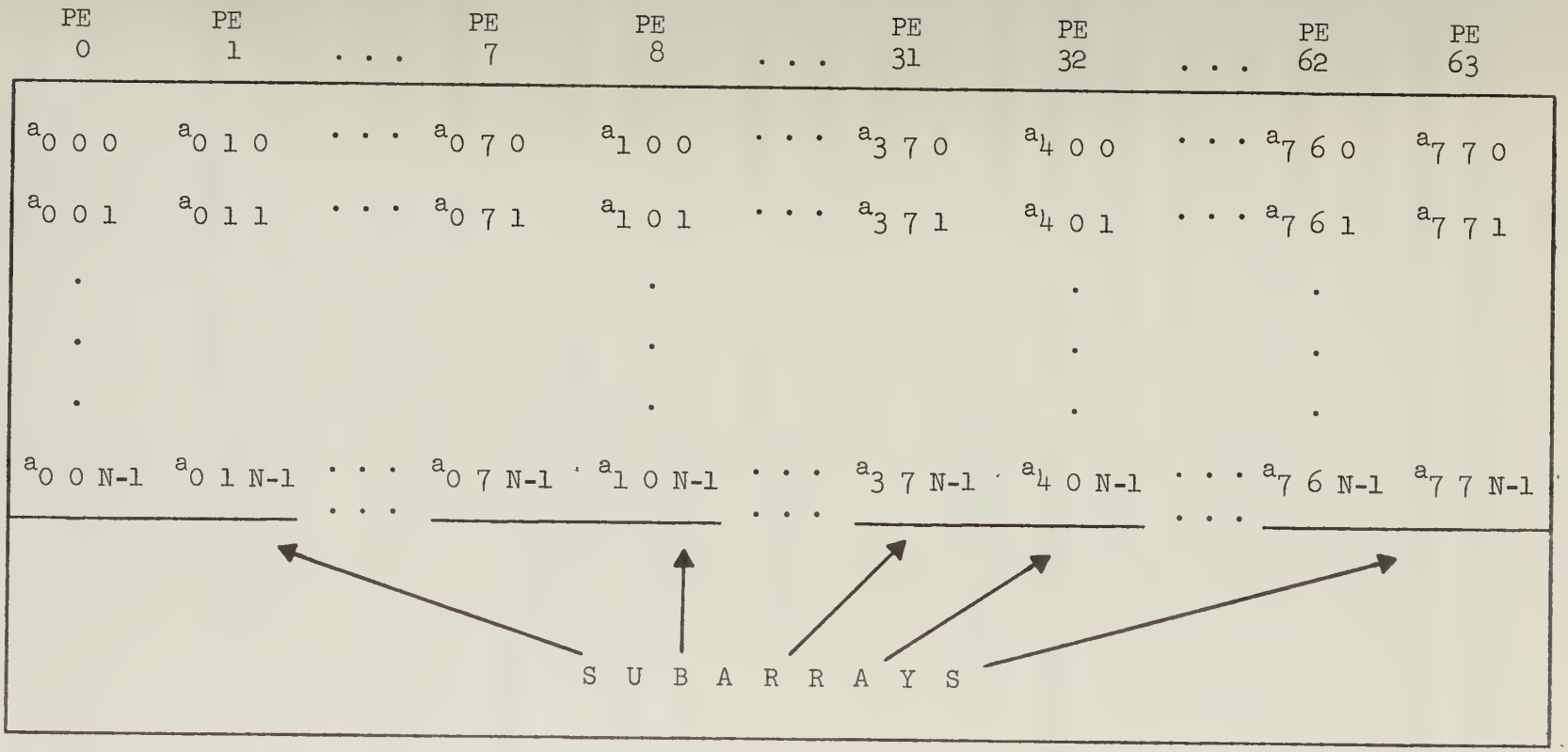


Figure 1. REAL ARRAY A[**0:7, **0:7, 0:N-1][+]

[+] See Ref. [4]

Three tables are provided specifically for the purpose of analyzing the structure of sets within an assignment statement. LHSTAB contains one entry for each group of sets that must be used together. LHSSETTAB contains link lists of the members in each of these groups. RANGTAB contains link lists of the value of SETREL [5] for those groups of sets which must be treated as a unit. This is only necessary for small subarrays as previously described. These tables are set up when the IL code is first entered in ARITHSTACK. RANGTAB is initialized to 0. Whenever a set linked variable is encountered, if the corresponding value of SETREL is N, then the N-th element of RANGTAB is tested. If the result is 0, then there have been no other occurrences of this set or other sets that must be varied with it, and a new link list in LHSSETTAB must be set up for this set as well as a new entry in LHSTAB. If it is nonzero, then there already exists a list of references to sets which must be varied with this set. RANGTABLHSLNK field of RANGTAB points to the element in LHSTAB for this group of sets. The field LHSSETLK in LHSTAB points to the element in LHSSETTAB that is the end of the list of sets. The new entry in LHSSETTAB which points to the occurrence of the set linked variable in ARITHSTACK is patched into the list with all pointers appropriately updated.

In forming these tables, information is entered in ARITHSTACK which provides some indication of how efficient it is for each occurrence of a set or a variable linked to it to vary that set across the PE's. This information consists of an estimate of the size of the set up to the number of PE's and its density. Whenever an operator has as one or

both of its operands a set linked variable, the results of that operation on the particular sets involved generates new efficiency ratings which are stored with the operator in ARITHSTACK. For example, take:

FOR (I,J) SIM ([1,2,...,100] × [1,2,...,50]);

A[I,J] ← B[3 × I,J];

Assume we are working in one quadrant, and both A and B are skewed arrays. The intermediate language code for this assignment statement will be:

SUBSCRIPTEDLEFTHANDSIDE

A

I

ENDOFSUBSCRIPT

J

ENDOFSUBSCRIPT

ENDOFSUBSCRIPTLIST

SUBSCRIPTEDPRIMARY

B

3

I

X

ENDOFSUBSCRIPT

J

ENDOFSUBSCRIPT

ENDOFSUBSCRIPTLIST

ENDOFARITHASSIGNMENT

When the second occurrence of I is entered in ARITHSTACK, a size of 64 will be entered and a density of 1. When the multiply sign is entered in ARITHSTACK, it will be given a set size of 64 and a set density of one-third. When the end of subscript operator is encountered, it is considered as operating on B and the result of the multiply expression and uses information about the array and the information stored with the multiply operator to update the rating in LHSTAB that gives the efficiency of doing that particular set group across the PE's. The ultimate result of this analysis, for our example, will be to vary J across the PE's.

2.5 Compiling Code From the Tables

The above mentioned tables are all built up by EXEC2 while interpreting the intermediate language for a single assignment statement. No code is generated during this process. The next step is to sort LHSTAB by the efficiency ratings. We wish to vary the least efficient sets in the outer loops and to compute all subexpressions in the outermost loop possible. The sort of LHSTAB provides the information necessary to achieve the former objective and LHSSETTAB and ARITHSTACK allow us to achieve the latter. For each entry in ARITHSTACK, three fields are provided. One field points to the operator operating on this entry. The other two fields point to the two operands of an operator and are 0 for operands. The compilation consists of N + 1 stages, where N is the number of independent groups of sets; i.e., the number of entries in LHSTAB. Each stage consists of two passes over part of ARITHSTACK.

A procedure called GETNEXT selects sequentially those elements of ARITHSTACK which can be compiled, given what loops have already been set up; i.e., what indicies are available. In our earlier example, as soon as values of I are available, we can compute $3 \times I$. The first pass of each stage determines optimal PE register usage, and the second pass finally generates code for all subexpressions that can be computed at this stage. The N + 1st stage completes the compilation of the statement.

3. OPTIMIZING SIMPLE ASSIGNMENT STATEMENTS

It is clearly impossible to efficiently compile a single short assignment statement for ILLIAC IV, but it is conceivable that a large number of simple assignment statements could be integrated into a fairly efficient ILLIAC IV program. Incorporating such a feature into a compiler presents two basic problems. The first is an algorithm for efficiently integrating a large number of interrelated assignment statements. Ordinarily the simple assignment statements will be scattered throughout the program. Also, many of the sequential calculations that are prime targets for an integration scheme are likely to be embedded as subexpressions in assignment statements containing SIM controlled variables. Filtering out and gathering together these candidates for the integration scheme constitutes the second problem.

Figure 2 contains a set of assignment statements and their associated tree. No node on this tree can be calculated until all nodes on subbranches have been calculated. The method of computing such a tree on ILLIAC IV involves first mapping assignment statements into PE's, in a more or less arbitrary manner. The assignment statements are restricted to a small number of operations like addition, subtraction, multiplication and division. ILLIAC IV can only perform one of these operations at a time. A count of the number of PE's that can take advantage of each of these operations is made and that operation which will be executed by the most PE's is the one that code is compiled for. Then the PE counts for all operations are revised and the process continues until all

$A \leftarrow B + C \times D$

$E \leftarrow L + B - C$

$F \leftarrow G + H \times I$

$K \leftarrow A + E + F$

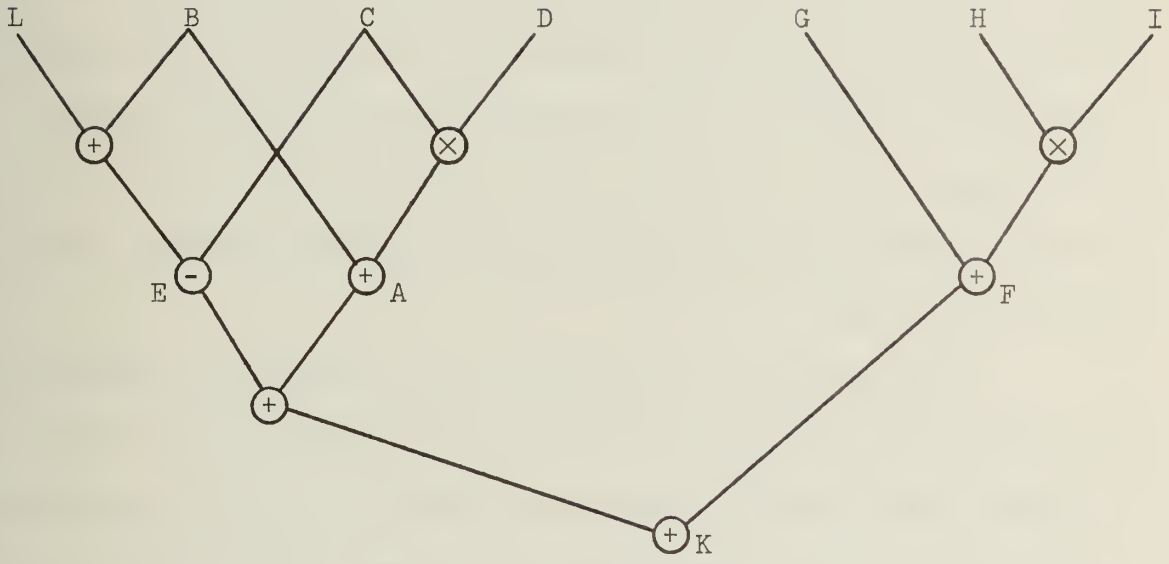


Figure 2. A set of interrelated assignment statements and their tree structure.

calculations have been performed. A similar algorithm is used to do routing to bring the results computed in one PE to the PE's where they are needed. This algorithm is invoked whenever the number of PE's eligible for any operation falls below a certain limit.

The problem of gathering together assignment statements for processing by this method is many faceted. What is desired is a rearrangement of the program where simple assignment statements, simple subexpressions, and simple expressions generated by the compiler, like address calculation, have been brought together at several collection points. To rearrange code in this manner requires an extensive analysis of the overall program to determine what subexpression and statements can be moved, and how far.

This analysis is carried out at the intermediate language level. The collection points are determined to be at the beginning of blocks, subexpressions are moved as physically high up in the code as possible, except that they are not moved past a block head unless they can be moved to the head of an outer block. The method produces a number of bonuses. Calculations inside loops tend to be moved outside when logically permissible. Thus, it is profitable to move nonsimple subexpressions also. Further, duplicate subexpressions can easily be eliminated because they tend to gather at the same point. Finally, for each block a record is made of what variables are nondynamic within that block. Thus, in Pass 2, any expressions generated using these variables can be added to the collection of subexpressions at the beginning of the appropriate block. At the head of this block, a transfer to the end of the block is compiled,

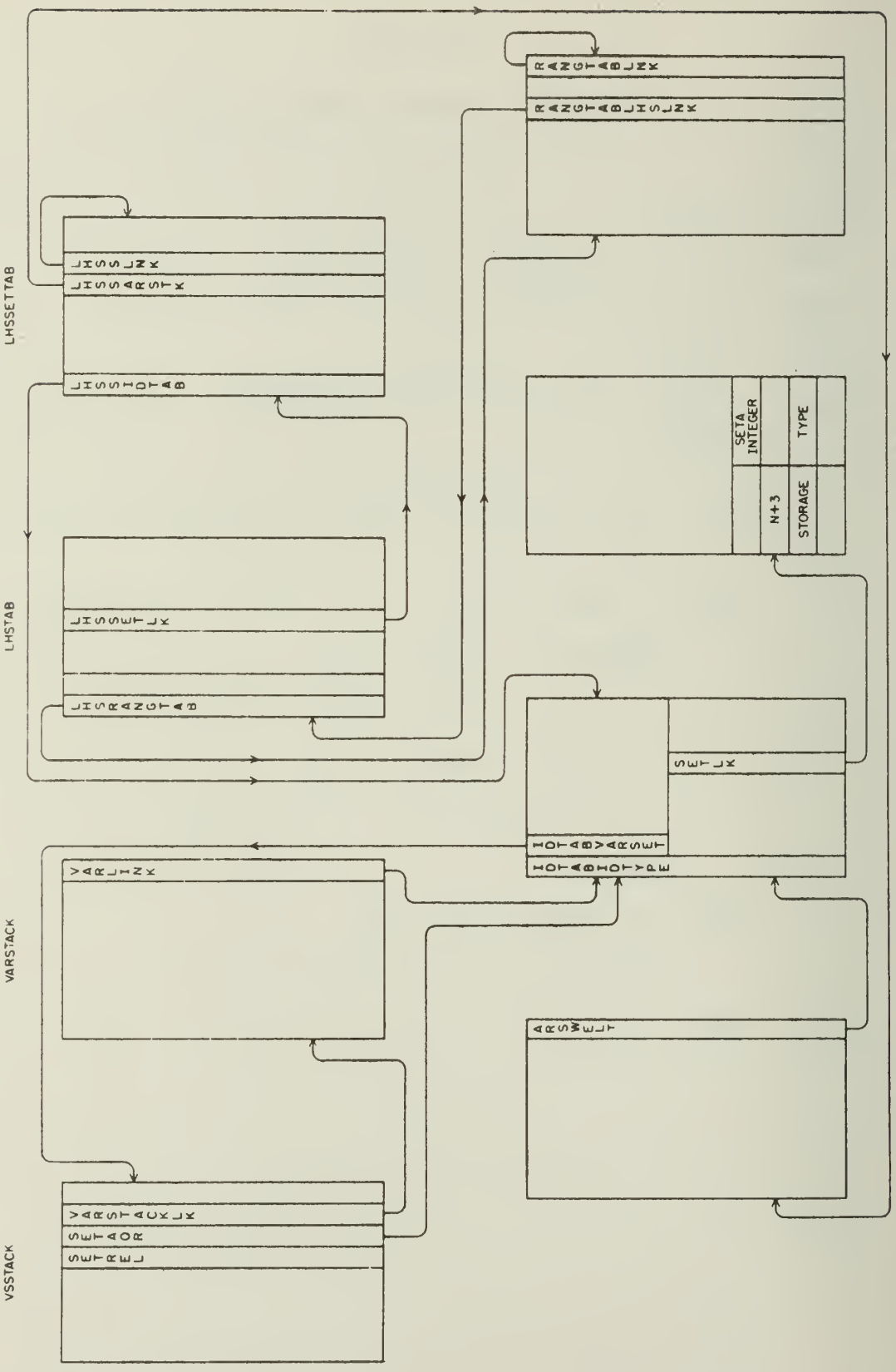
and when all code in the body of the block has been generated, the complete collection of assignment statements is compiled followed by a transfer back to the beginning of the block.

4. CONCLUSION

The problem of generating efficient code for ILLIAC IV from a high level language similar to ALGOL requires considerably more analysis of each assignment statement than is necessary for conventional machines. Predictions about the overall efficiency of the resulting code are more difficult to make. Nonetheless, algorithms have been devised which work reasonably well on a very large and general class of problems, and these should allow a programmer to write reasonable code with a minimal knowledge of ILLIAC IV and the details of TRANQUIL.

APPENDIX: A SAMPLE TRANQUIL PROGRAM

```
BEGIN
REAL SKEWED ARRAY A, B[0:100, 0:100];
INCSET JJ;
MONOSET II(1) [27:6], KK(1) [100:100];
INTEGER I, J, K;
II ← [2, 10, 13, 15, 21, 24];
JJ ← [2, 4, ..., 98];
FOR (I) SEQ (II) DO
     BEGIN FOR (J) SIM (JJ) DO
               KK ← SET (J:A[I,J] < B[J,I]);
          FOR (K) SIM (KK) DO
               A[I,K] ← A[I + 1,K] + B[I,K + 1]
     END;
FOR (I,K) SIM (II × KK) DO
     A[I,K] ← A[I + 1,K] + B[I,K + 1]
END
```


RANGTAB
RANGTABJNK
RANGTABJHSSJNK
LHSSETTAB
LHSSSJNK
LHSSSARSTK
LHSSIDTAB
SETTAB
SETA INTEGER
N+3
STORAGE
TYPE
LHSTAB
LHSSETJK
LHSRANGTAB
IDTAB
SETJK
IDTABVARSET
IDTABIDTYPE
VARSTACK
VARJNK
ARSWEJT
VSSTACK
VARSTACKJK
SETADR
SETREJ
ARITHSTACK

LIST OF REFERENCES

[1] Wilhelmson, Robert Bernard, "Control Statement Syntax and Semantics of a Language for Parallel Processors," M. S. Thesis in Computer Science, Section 2.2.4, p. 20, January 1969.

[2] Trout, Robert, "A BNF Like Language for the Descriptor of Syntax Directed Compilers," M. S. Thesis in Computer Science, January 1969.

[3] Wilhelmson, Robert Bernard, "Control Statement Syntax and Semantics of a Language for Parallel Processors," M. S. Thesis in Computer Science, Section 3.3.6, p. 60, January 1969.

[4] Muraoka, Yoichi, "Storage Allocation Algorithms in the Tranquil Compiler," M. S. Thesis in Computer Science, p. 32, January 1969.

[5] Wilhelmson, Robert Bernard, "Control Statement Syntax and Semantics of a Language for Parallel Processors, " M. S. Thesis in Computer Science, Section 3.3.5, p. 59, January 1969.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Department of Computer Science University of Illinois Urbana, Illinois 61801	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

TRANQUIL ARITHMETIC

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Research Report

5. AUTHOR(S) (First name, middle initial, last name)

Paul Peter Budnik

6. REPORT DATE January 6, 1969	7a. TOTAL NO. OF PAGES 29	7b. NO. OF REFS 4
8a. CONTRACT OR GRANT NO. 46-26-15-305 b. PROJECT NO. USAF 30(602)4144 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) DCS Report No. 296 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DCS.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Rome Air Development Center Griffiss Air Force Base Rome, New York 13440

13. ABSTRACT

The assignment statements allowed in Tranquil are described with emphasis placed on writing efficient code. A description of the implementation of the assignment statement part of the compiler is given in considerable detail and is likely to be of interest mainly to those who find it necessary to correct, alter or make additions to the Tranquil Compiler.

DD FORM 1 NOV 68 1473

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Tranquil Arithmetic						
Set Assignment Statements						
Boolean Assignment Statements						

JUN 4 3 1973

BOUND BY HNM

UNIVERSITY OF ILLINOIS-URBANA
3 0112 045402051